CHAPTER P-5

Conceptual Twining

In our fifth practical chapter, we turn from the technical to the conceptual. In previous practicals, we built up a repertoire of capabilities. You learned to do increasingly sophisticated things with Twine: linked storytelling, textual variation, formulaic text generation, and inclusion of other media. We experimented with Harlowe, SugarCube, and JavaScript. This chapter, which will rely exclusively on Chapbook, will mainly use coding techniques we have already presented, so it serves at least partly as a capstone, reviewing and consolidating the work so far. Our emphasis here is on using Twine in the service of various schemes and ideas—what we call *conceptual Twining*.

The word *concept* can be hard to define with precision and easy to toss around loosely. "Reality—what a concept!" as Robin Williams's Mork used to say. For the purposes of this chapter, *concept* refers to an expression or design that operates simultaneously on two levels. One of these is direct or literal. In our five examples, we will build more or less familiar Twine texts: fragmentary stories, little games, riddles. Aside from whatever dubious charm these texts may have in themselves, they invite further reflection on Twine and its uses and perhaps on stories, games, and language generally.

◊ We continue the typographic conventions of earlier practical chapters, where invitation/instructions to type in code fragments are boxed and marked with the symbol you see here. You can, of course, skip either the typing or the intervening explanations, though, in a chapter dedicated to concept, those explanatory bits are especially important.

Supporting materials for this chapter can be found online at https://github.com/AMSUCF/Twining. See our discussion at the beginning of chapter P-1 about using the .html and .txt files to follow along or adapt our code to your own purposes.

Example 5.1: *Labyrinth*

This project has a history. In 1991, Stuart wrote *Victory Garden*, a long-form hypertext fiction developed with an authoring tool called Storyspace, which was then under development by Jay David Bolter and Michael Joyce. In those days, many fewer people knew about the internet (then called "Internet" with a big *I*), and the World Wide Web was, no kidding, *an application program*, what we would today call a web browser—only nobody in those days ever said, "the Web," with or without capital letters. Like text adventures a decade or so earlier, hypertext was a fresh concept. No one knew exactly what could be done with it, though a few people had ideas.

The conception for *Victory Garden* was a big, tangled mass of stories intersecting at various points. This was by no means an original invention, as anyone up on their George Eliot, John Dos Passos, or Richard Linklater will tell you, but its application to digital media up to then had belonged mainly to parser-based games. With all respect to that form, some of us, including Michael Joyce and Judy Malloy, wanted to try something different—*a story that would change each time you read it*, as Michael said—shifting the stress from game to story.

Given that emphasis, readers needed to enter the narrative thicket as unpredictably as possible, but the early version of Storyspace made this difficult—it had no capacity to choose a passage at random. With

randomness unavailable, the author reckoned the next best thing might be confusion, so he designed an elaborate set of passages with bifold links—a verbal labyrinth—in which it was hoped readers might productively wander toward the main events, discovering various predefined ways to get there.

To make this process of discovery more meaningful, the labyrinth asked participants to build a sentence one word or phrase at a time. At each place in the story, readers chose between two candidates for the next word in the sentence. The succeeding place repeated the sentence as it currently stood and either offered another pair of choices or came to a conclusion. That conclusion was a complete sentence that somehow stamped the reader's ticket for the ride that followed.

Three decades is a long time, in which interactive storytelling has made considerable progress, but we think the old labyrinth concept remains useful. Our first project in this chapter implements the labyrinth in Twine.

◊ Start a Twine story, change the story format to Chapbook if necessary, and give it any title you like, though *Labyrinth* seems an obvious suggestion. Change the name of the default first passage to "Origin" and enter the following text:

```
In the

>[[green bag]]
>[[vicinity of metaphor]]
```

Close this passage and have a look at your structure view. You'll see two linked passages, both so far unwritten. We'll get to one of them shortly, but for the moment, a note about those right angle brackets (greater-than signs) at the left of each link. In Chapbook, these symbols identify a *fork*, a choice of two or more options that are set off typographically when the passage is displayed.

To keep this chapter from growing tedious, we'll only explore one branch from the initial prompt—only one course of the labyrinth, in

technical terms. You are free to develop the rest of the maze in your own way. The online materials include a completed version of this project with all possibilities written out, and we'll discuss a few before we're done. For the moment, let's pursue the green bag.

◊ In the passage labeled "green bag," type the following:

```
In the green bag

>[[we found]]
>[[it was possible]]
```

As you can see, each phrase plausibly extends this rather strange sentence, and at the start of each passage, we display the sentence as currently composed. The fork construction gives us another two links and another pair of auto-generated passages. Like spectators at a magic show, we applaud the Amazing Klimas every time this trick comes off. We'll go with the first one, "we found."

◊ In the passage labeled "we found," type the following:

```
In the green bag we found

>[[the recipe for Detroit]]
>[[a rich deposit of language]]
```

Once again, we have a fork and a pair of resulting passages—though don't worry, we're nearly at the end of this line. This time, we'll choose the top bunk.

◊ In the passage labeled "the recipe for Detroit," type the following:

```
In the green bag we found the recipe for Detroit
```

What follows from this declaration is negotiable. We have a suggestion, but you can do differently if you like. You'll note we haven't added terminal punctuation to our nearly finished sentence. You could put in a period and then write more sentences to follow, making this the first full passage of a branching story in prose. Something like this: "In the green bag we found the recipe for Detroit. 'Put that back this instant!' the Chief Investigator thundered. Which we did, but not before committing the contents to memory."

Links out of such a passage, or a different passage of your own invention, would then be up to you. Before we wrap this project up, we'll mention a variant outcome that turns the labyrinth concept to another purpose. What if the completed sentence functions not as the first line of a story but as the title of a poem? All the other possible outcomes from the labyrinth might work likewise so that the word-maze would function as a hypertextual framework for a collection of poems. There's a lovely coincidence here: in the print tradition, such a collection is often called a *chapbook*. Here's our version of the recipe for Detroit:

cars the size of cars
implying
a continent of erasure
in the scattering of a people
plus two ideas,
unpromising,
and the given name of an advertised lawyer
featuring seven types of ambiguity,
overlooking a sunrise, and
one sad invention
with statistics,
disaggregated,
with a warm place to put the results.

Poem notwithstanding, here is the structure map of our verbal maze:

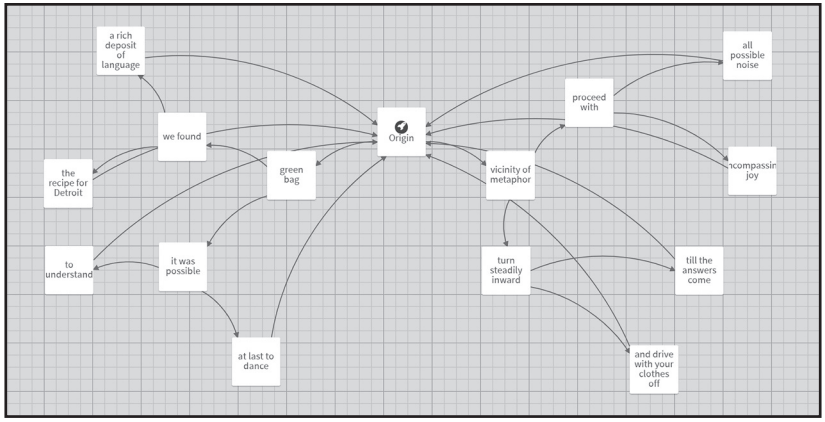

Figure 23: Structure map of the *Labyrinth* project

As you can see, our labyrinth offers seven other sentence trajectories:

In the green bag we found a rich deposit of language
In the green bag it was possible to understand
In the green bag it was possible at last to dance
In the vicinity of metaphor turn steadily inward till the answers come
In the vicinity of metaphor turn steadily inward and drive with your clothes off
In the vicinity of metaphor proceed with all possible noise
In the vicinity of metaphor proceed with encompassing joy

In the completed version, there's a bit of free verse or prose poetry at the end of each one. We have used this labyrinthine model in various classes over the years, and our students have come up with many interesting applications of the stepwise-approach concept: as introductions for other sorts of creative work, such as video or music; as a test of attitudes or preferences; and of course, the inevitable guessing game with one or more ostensibly right answers. Every game a riddle, as we'll see in our final example. Meanwhile, let's continue with overtures and opening acts.

Example 5.2: *Spooky*

This example bears a certain resemblance to the first in that it, too, is designed as a preamble or entry point for further fiction. This one works more in generalities or atmosphere instead of the specificity of a title or first line. The concept here is mood-setting, foreshadowing, or coalescence—a distillation or inspiration of horror. In addition to reviewing our familiar method of randomized text selection, this example also demonstrates an approach to page layout that works around some basic limitations of Chapbook. No art without constraints.

The first step of this exercise can take place either in or out of Twine, as you like. You need a list of words and phrases that might either come from or be associated with a tale of horror. If you've read Mary Shelley or Poe or Lovecraft or played *Call of Cthulhu*, this will be a cinch. You could write your list on paper, in a convenient word processing document, or in a Twine passage. If you're going to do the last, perform this first step before writing.

◊ Open a new story and name it *Spooky*. If you're using the default passage to write your list, you may want to name it "List." Otherwise, name your first passage "wordcloud."

Write the list as previously described. The length is up to you. A set of ten items is probably minimal; twenty-five feels like more than enough.

As you have probably guessed, we'll use that word list as a database for random selection with the substitution grammar technique we demonstrated in chapters P-2 and P-3. It should be very familiar by now.

◊ If you didn't designate or set up a passage called "wordcloud" in the previous step, do that now. Open that passage and type the following:

```
words: []
--
```

We've started a variables section for the "wordcloud" passage, and in that section, we have declared an array variable called "words." Fill in (the technical term is *populate*) that array with the words and phrases in your list, being careful to enclose each one in single quotation marks, separated by commas. There's no comma following the last item. For reference, here's a selection from our version of the array:

```
'groan','splatter','apparition','testament','Ab
igail','unliving','rupture','invasion','ovuloid
','pulsing mass','pullulating','unquiet','blood-
chilling','viscera','abomination','immense','b
eyond the veil','partly decomposed','unnatural
fusion','funebrous','rotting fruit','spoiled mea
t','shambling','shade','axehead','severing','unr
easoning','revenant','shrieks'
```

◊ Now we'll add one more line to our variables section, immediately below the definition of the array:

```
theLink: words[Math.floor(random.fraction*words.
length)]
```

Be careful to leave in place the two dashes that close the variables section.

You've seen this code before, but it's sufficiently complicated to make a review worthwhile. We're declaring a variable, *theWord*, to hold a selection from the *words* array. Selections from an array are made by putting a number into [square brackets]. The complicated expression inside the brackets will be processed into a convenient number—you'll remember the process. The *random.fraction* expression is from Chapbook. It returns a decimal number between zero and one. Multiplying that number by the length of the array gives us another decimal greater than zero. We use the *floor* method of the JavaScript *Math* object to round this fraction down to the nearest integer, which will be between

zero and the length of the array minus one. All arrays are numbered beginning with zero, so the last item has a number one less than the overall length. That's what we mean by convenient number.

◊ Now that we have *theLink*, let's serve it up. After the two dashes closing the variables section, type the following:

```
{theLink}
```

This insert calls on Twine to display whatever value is stored in "theLink."

Make sure "wordcloud" is identified as the start point of your story, then run or test your project several times. You should see one word or phrase from your shop of horrors every time you restart. Since nothing prevents repetition, you may see the same selection on successive tests, so be patient. The more words and phrases in your array, the lower the chance any item will repeat—though larger sets increase your exposure to mistyping.

In case of repetition, we generally try up to five times before assuming an error. If you see nothing on-screen, check to be sure you entered the line in the previous step properly, using curly braces, not square brackets. Assuming no mistake there, check the definition of your array. It's very easy to forget a quotation mark or comma.

This might be a good time to reread our section on debugging in chapter P-2. If you're struggling to find the problem, copy the contents of your array to a temporary passage and replace them with one or two test words. Once you've found the bug, or if your code was fine in the first place, you are ready to proceed.

This project is designed to do a bit more than display one word or phrase at a time. If you look at the completed version in the online materials, you'll see it presents a scattering of words across the screen, each a live link. Clicking on any of the horror words refreshes the screen with a new set of links, likely at different positions, the better to emphasize the randomness of the effect.

Making all this happen requires more than what we currently have in the "wordcloud" passage. Let's start by getting just one of these horrifying links to appear.

◊ In the text body of "wordcloud"—the part below the double dashes—take out the line we added in the previous step and replace it with this:

```
[[{theLink}->wordcloud]]
```

This line has a lot going on, typographically. It starts with double square brackets, the fundamental Twine convention for creating a hypertext link. Inside the brackets, there's a set of curly braces around the name of our random-phrase variable, "theLink," and then the familiar basic syntax of a stylized arrow -> pointing to the destination passage "wordcloud." Yes, that means this link reloads the passage that contains it, a trick we've seen previously.

Now it's time to test or run. You should see a single word or phrase from your array as before, but this time hyperlinked. Clicking on the link should change what is displayed, though remember our earlier advice about possible repetition. If you find errors, use the procedure we recommended to root them out. If all is well, proceed.

So far, we haven't done anything to introduce the scattered or floaty design aesthetic that appears in the finished version. We'll add those pieces now.

◊ Edit the text body of "wordcloud," adding lines above and below the single line that is there presently, ending up with this:

```
<br /><br />
<div style="text-align: 5em">
[[{theLink}->wordcloud]]
</div>
```

If what you just typed looks like part of a web page, congratulations, you know something about HTML, the basic construction kit of the World Wide Web. More people should be like you. We've pointed out that Twine is made of JavaScript, which is also part of web infrastructure. It's the scripting language that extends the functionality of web pages, and Twine stories are delivered in that form. Just as we can slip JavaScript expressions into Chapbook en passant, we can also turn to HTML when needed. The first HTML effect we use here is the line break tag <*br* />. Two of these tags in sequence, as we have here, create a skipped line or vertical space on the screen.

Next, we introduce a division, or *DIV*, using the paired tags or *container* <*div*></*div*>. A DIV is a block-level element of a web page—or in this case, Twine passage—one that is set off from previous elements as paragraphs are. (We could also have used the *P* or paragraph container here.) Notice the tag that introduces our DIV. It contains some added information:

```
<div style='text-indent: 5em'>
```

This tag calls on the third major element of web technology, CSS. That invocation happens in the expression *style*=. A style sheet, technically an *inline style sheet*, in this case, modifies the appearance of the DIV to which it is attached. Here we call for that element to be indented five ems from the left margin. If you're a graphic designer or typographer, you know an em is the width of an em dash, like this one—which varies with the font family and size being used. Web designers prefer these relative measurements nowadays. For our purposes, let's just say it's a unit of horizontal space. (It's a game of ems, not inches.)

If you run and test now, you should see your randomly selected link offset from both the top and left of the window. This is part of the effect we're aiming at, but the full concept involves multiple, randomly chosen links, placed differently on the screen at each reload. We can get this to happen with the tools at our disposal—HTML, CSS, and

Chapbook—but we'll need to blend them into a slightly more complicated cocktail. Let's handle the vertical spacing first, as that's the more familiar part of the recipe.

◊ Once again edit the text body of "wordcloud." Replace the pair of *
* tags with what you see here. Leave everything else alone.

```
{vSpace}
<div style="text-align: 5em">
[[{theLink}->wordcloud]]
</div>
```

You've just replaced those *
* tags with a variable that doesn't yet exist, so let's take care of that. In your variables section, below the line that defines *theLink*, add the following:

```
breakers: ['<br />','<br /><br />','<br /><br
/><br />']
vSpace: breakers[Math.floor(random.
fraction*breakers.length)]
```

The *breakers* array contains three collections of *
* tags, which when applied will set our DIV at variable distances from any element above it or from the top of the screen. The *vSpace* variable contains one of these collections (or strings). If you test your project at this point, you should see the randomized link appearing at a variety of vertical positions.

We could use the same array-selector strategy for the initial DIV tag, using different values for the em spacing, but this would involve a whole lot of repetitive typing. The only thing we really need to change is the number that precedes "em." We achieve this effect by using a variable for vertical spacing and using the Chapbook *random* function when we generate that variable. Here's the way it works:

◊ In the variables section of "wordcloud," below the material you added in the previous step, type in the following:

```
hSpace: "<div style='text-indent:" + random.d10
+ "em'>"
```

Pay superclose attention to the placement of single and double quotation marks in this line—we need to use both. We take the familiar inline style sheet and cut it apart where the number occurs. Instead of that number, we insert the expression *random.d10*, which generates an integer between 1 and 10, inclusive. Notice we need the string 'em' and the closing angle bracket at the end. All this is enclosed in a set of double quotation marks because it is a string, a sequence of words and numbers treated as text. When we invoke this string variable, it is added to the code of our passage and treated by the browser as an HTML expression, which is how browsers treat any text containing angle brackets.

If you test your project now, your randomized link should appear at unpredictable locations both horizontally and vertically. At this point, we've built the core pieces of the project, but there are still important elements missing. So far, we only have one link in play, but we want several. We'll handle that feature next.

It might occur to you that all we need to do in order to include more links is duplicate the code that brings in the first link. That's partly right, but we need to adjust things a bit because of a certain feature of Chapbook. That story format allows us to define variables only once, when the passage loads into memory. So if we use our *hSpace* and *vSpace* variables for additional links, they'll have the same vertical and horizontal offsets as the first one. That's not what we want. Here's the work-around, beginning with the vertical spacing:

◊ In the variables section of "wordcloud," find the section that handles vertical spacing. You can leave the line that defines the *breakers* array as it is, but replace the line that follows, the one that defines *vSpace*, with the following:

```
vSpace1: breakers[Math.floor(random.
fraction*breakers.length)]
vSpace2: breakers[Math.floor(random.
fraction*breakers.length)]
vSpace3: breakers[Math.floor(random.
fraction*breakers.length)]
```

You can copy the original definition of *vSpace* and paste it in three times to define its new companions. Be sure to add the numerals that make these three variables individual.

This new code creates a trio of variables, each with a selection from the *breakers* array. Two or more of them may have the same selection, but that's not a problem for this project. We only have room for a limited amount of vertical spacing, anyway. Next, we'll handle horizontal spacing:

◊ Replace the line that currently defines *hSpace* with this set:

```
hSpace1: "<div style='text-indent:" + random.d10
+ "em'>"
hSpace2: "<div style='text-indent:" + random.d10
+ "em'>"
hSpace3: "<div style='text-indent:" + random.d10
+ "em'>"
```

Copying and pasting will work here as well, since the only difference in these lines, as in the vertical spacing, is the numeral that makes each horizontal spacer unique.

In order to support our scheme, we'll also need three servings of our link text. By now, the procedure should be familiar:

◊ Replace the line that currently defines *theLink* with this threesome:

```
theLink1: words[Math.floor(random.fraction*words.
length)]
theLink2: words[Math.floor(random.fraction*words.
length)]
theLink3: words[Math.floor(random.fraction*words.
length)]
```

Copying and pasting is fine, but once again, be sure you've added the numbers to the variable names. No other changes are necessary—we're making three unique selections from our *words* array, using the same procedure each time, but with potentially a different random selector.

Now we need to adjust the contents of the text body in "wordcloud" to support three floating-horror links.

◊ Again, remember that we're in the text body this time and not the variables section. The code that currently conjures up our solo link looks like this:

```
{vSpace}
{hSpace}
[[{theLink}->wordcloud]]
</div>
```

Veteran web coders may find this a bit disturbing because having a closing tag like * * without an initial * * tag is ordinarily an error. However, that initial tag is loaded into the *hSpace* variable, so we're fine.

If we want three links, all we need to do is copy and paste our link construction two times, making necessary adjustments:

```
{vSpace1}
```

```
{hSpace1}
[[{theLink1}->wordcloud]]
</div>

{vSpace2}
{hSpace2}
[[{theLink2}->wordcloud]]
</div>

{vSpace3}
{hSpace3}
[[{theLink3}->wordcloud]]
</div>
```

Do not forget to change all three variable invocations—for vertical spacing, horizontal spacing, and link text—in each of the three segments. The numbers in the variable names are crucial.

You can test at this point. You should see three live links, randomly planted on your screen, in an arrangement that changes each time you reload. Clicking any of the links will cause a reload. If that's what you're seeing, you are ready to proceed to the final stage of construction: offering the reader a chance to leave the horrifying word cloud and enter the story proper. We'll specify that we may not make this offer every time the passage loads, and if we do offer the exit, we'll do it only once per iteration. We'll build these features in four steps. The two in the middle are a bit detailed, but the last is very simple, as is the first:

◊ Let's add one final asset to our variables section. This variable could be introduced anywhere in the section, but we'll put it below everything we currently have. Add this line:

```
exitPos: random.d4
```

As you'll remember, *random.d4* returns a value of 1, 2, 3, or 4. We actually only need a range of 1 to 3, but that would require a bit more tedious typing. Using the virtual d4 slightly reduces the chances of our exit link appearing, but this won't be a problem.

If you want to improve the odds for the exit link, you could use this instruction instead:

```
exitPos: Math.ceil(random.fraction*3)
```

The *exitPos* variable determines which of our three horror links will be replaced by a link to the main story. Remember, though, that we want the possibility of the escape link not appearing in any given iteration. Doing that will involve one last piece of detailed coding, described in our second step:

◊ In the main text body of the "wordcloud" passage, find the first of the three link units. Delete it and replace it with the following:

```
{vSpace1}
{hSpace1}
[if exitPos===1 && random.coinFlip]
[[Begin]]
[else]
[[{theLink1}->wordcloud]]
</div>
[continued]
```

The first two lines in this section are the same as in the previous version—we're still invoking the strings that create randomized horizontal and vertical spacing. After that, there are some changes. We introduce an *if* condition that makes the display of our link to the "Begin" passage depend on two factors: *exitPos* being 1, and the value of *random.coinFlip* being true. There is a one in four chance of the former condition

(if we use d4) and a one in two chance of the latter. That means a one in eight chance our first link will be replaced with the story exit.

After the standard link to the "Begin" passage, there's an *else* statement, after which the rest of this code chunk is as it was in the beginning, except for the *[continued]* statement at the very end. This statement terminates the *if/else* logic and lets us treat the next link segment independently.

You might want to test at this point. Keep trying until you see your first link replaced by the link to "Begin." If that doesn't happen after about ten attempts, you probably have an error. Hopefully, this procedure will go fine, and you can proceed to the third and penultimate step:

◊ Modify the remaining two link segments to have a similar structure to the first, though remember to change the crucial numbers—the value of *exitPos* and the numbers of the spacer variables—as indicated. Here's what you should have:

```
{vSpace2}
{hSpace2}
[if exitPos===2 && random.coinFlip]
[[Begin]]
[else]
[[{theLink2}->wordcloud]]
</div>
[continued]

{vSpace3}
{hSpace3}
[if exitPos===3 && random.coinFlip]
[[Begin]]
[else]
[[{theLink3}->wordcloud]]
</div>
```

You don't need to include *[continued]* after the third option, as nothing follows it.

Now there's just one easy thing to do, which we won't bother writing out as a formal instruction. Go back to your structure map and find the new passage called "Begin." Write something unspeakably spooky there. For testing purposes, you might also want a link back to "wordcloud."

And so our maleficent mechanism is complete. On a technical level, this project shows how to weave semi-intricate cobwebs of code, grafting disembodied bits of HTML into Chapbook with diabolical abandon. One code-packed passage ("wordcloud") can support endless cycles and iterations or as many as your reader can stand. Conceptually, this example builds on the tension between anticipation and action—a main component of horror—or between a fitful, recursive flow of possibilities and the forward progress of conventional storytelling.

Like the labyrinth of our first example, the evocative links of example 5.2 could be applied to other genres and purposes besides the one suggested here. The links in our example are based on a somewhat arbitrary collection of words, but as we hinted, they might be drawn directly from the text they precede. Readers in the waiting room would thus encounter hints and teasers for what awaits beyond the entry. Taking this concept further could bring us back to the original motivation for the labyrinth in *Victory Garden*—randomized beginnings. Suppose each of those brief quotations from the work contained a live link to the passage where its word or phrase occurs. In a sizeable work, there could be a large constellation of starting points. If that kind of sprawl is not desirable, various prefatory links might cluster around a more limited number of options. The alternation between link-scattering and reading ahead might also be used between sections of a multipart work. Possibilities abound. The horror . . .

Example 5.3: *Active Measures*

Let's start this one with a disclaimer: we have nothing against text adventures. Both authors of this book have been significantly influenced by games of this kind, one of us before the advent of computer graphics, the other afterward. We belong to communities like ELO and IFTF,

where turn-based, procedural stories are treated with love and respect. The Inform programming language, to which we have frequently referred in this book, means much to us. One of us, no kidding, has been known to dream in it.

However, as we pointed out all the way back in chapter P-1, there are two ways to think about procedural storytelling. One approach favors procedure, the other story. You will recall we discussed contrasting ways to treat links in Twine, either by separating them formally from the narrative, in the manner of interactive fictions and game books, or by planting the links in a single narrative stream, as in hypertext fiction. The distinction is largely arbitrary, and many writers do both, but this next exercise pits one mode against the other. It's a concept.

◊ Start a Twine story, remembering to use the Chapbook story format, and call it *Active Measures*, or what you will. Change the name of the default first passage to "Action!" Yes, Twine allows exclamation points in passage names, and you can name the passage something else if you like, provided you make relevant changes as we go. Enter the following text:

```
verbs: ['Take','Drop','Examine','Eat']
theVerb: verbs[Math.floor(random.fraction*verbs.
length)]
--
```

Here it is again, our inevitable grammar of substitution. Do we know how to do anything else with code? Maybe—see the bonus practical chapter in the appendices. Do we *want* to do anything else? Not all that much, apparently.

Note that we're working on the variables section of our passage, that's what those two dashes on the final line indicate. The topmost line of this code chunk declares an array, which in this case is a comma-separated list of words (strings) in a specific sequence. The second line defines a variable called *theVerb* and assigns it a randomly chosen item from the *verbs* array. But you knew that!

There's quite a bit more to do with "Action!," but we should first say a few things about the design of this project. The "Action!" passage will hold data and logic we'll use to generate randomized phrases on demand—again, something we've done in many previous exercises. There will be two other passages in this project, one calling back to "Action!" in a self-perpetuating loop, and the other representing the way out of that loop: the same pattern we used in the previous example. We'll come to these passages eventually. Meanwhile, let's build more of the "Action!" passage. It's pretty extensive.

◊ Enter three blank lines ahead of the double dashes. Move your cursor up one line and type in the following. There's a lot of typing here. If you want to shorten any of the arrays, feel free. You could also expand them without causing any problems.

```
nouns: ['eyedropper','skillet','cleaning
robot','fishbowl','phrasebook']
theNoun: nouns[Math.floor(random.fraction*nouns.
length)]

IVerbs: ['Jump','Cry','Scream','Wait','Pass','Br
eathe','Exist','Persist','Think']
theIVerb: IVerbs[Math.floor(random.
fraction*IVerbs.length)]

directions: ['north','east','south','west','back
','forward','up','down','nowhere','anywhere','si
deways','to pieces']
theDir: directions[Math.floor(random.
fraction*directions.length)]

askTell: ['Ask','Tell','Notify','Enlighten','In
form']
theAskTell: askTell[Math.floor(random.
fraction*askTell.length)]
```

```
persons: ['Mr. Jones','Starbird','Flux Man','Jim
my','Otto','Maisie','Hermione']
thePerson: persons[Math.floor(random.
fraction*persons.length)]

topics: ['astrometry','fine wines','outwitting
the Troll','contents of the box','the
key','stuff']
theTopic: topics[Math.floor(random.
fraction*topics.length)]
```

As always, you do not need to put in these lines exactly as they appear. You can (should!) substitute your own words in any of these lists, provided yours are of the kind that is called for—singular nouns in the *nouns* array, proper names in the *persons* array, and so forth. As indicated, you can delete or add items. Because we use the *length* parameter of the array, you can change the size of the array without breaking the code. Do be careful to use single quotes around every word and make sure the commas go outside of the quotation marks. In the selection lines, the ones that invoke the *Math* object, pay close attention to the succession of parentheses and square brackets.

You are building the raw materials for several generated sentences or phrases. The next chunk of code contains the templates for those phrases.

◊ Still within the "Action!" passage, below the last line you typed before, enter the following:

```
phrase1: theVerb + ' the ' + theNoun
phrase2: theIVerb
phrase3: "Go " + theDir
phrase4: theAskTell + " " + thePerson + " about
" + theTopic
phrase5: "Give the " + theNoun + " to " +
thePerson
```

```
phrase6: "Take the " + theNoun + " from " +
thePerson
```

There are six variables, each containing a phrase generated from the arrays and variables you previously defined. As you can see, five of these phrases are multiword combinations. Be careful to type spaces where they are called for, around words like *to* and *from*. Notice that *phrase2* simply invokes the premade selection from *IVerbs*. That's because this phrase consists of a single word, an intransitive verb like *Jump* or *Wait*.

That was quite a lot of detail, though we're only about halfway done with "Action!" Don't worry, the rest of the project is less verbose. After resting eyes, wrists, and fingers, once more unto the breach.

◊ Still within the "Action!" passage, after the line that defines *phrase6*, make a new line and type the following:

```
theRoll: Math.ceil(random.fraction*6)
```

This line generates a random number between 1 and 6. Chapbook offers a perfectly good way to do this—*random.d6*—but we'll eventually want to roll a seven-sided die, which Chapbook does not provide for directly. That's why we're using *random.fraction* rounded up with the JavaScript *Math* object. Now one more push to finish the "Action!" passage.

◊ Find the double dashes that mark the end of your variables section. If you inadvertently took them out at some point, put them back. Below those dashes, type the following:

```
[if theRoll===1]
[[{phrase1}->Scene]]
[if theRoll===2]
[[{phrase2}->Scene]]
[if theRoll===3]
[[{phrase3}->Scene]]
[if theRoll===4]
```

```
[[{phrase4}->Scene]]
[if theRoll===5]
[[{phrase5}->Scene]]
[if theRoll===6]
[[{phrase6}->Scene]]
```

If you can already tell what these lines do, congratulations—you're a Twine master! If they're a mystery, read on. What we have here is a sixfold chain of *if* conditions, tracking the possible values of our virtual dice roll. You may recall that in Chapbook, the *if* condition may be used only outside of the variables section and only to control the display of text. We're meeting both requirements here, though the text we're displaying—contents of one of our phrase variables—will show up within a passage other than the one we're working on here. Don't freak, we'll explain that in a bit.

Meanwhile, a bit more detail of the step you just completed. The line following each of our *if* conditions contains a hyperlink whose verbal content is one of our variables. You saw this design pattern in the previous example. Each of these links goes to the "Scene" passage. It's time to write that passage.

◊ The passage called "Scene" will be added to your structure as soon as you close the "Action!" passage. Open "Scene" and type what you see here:

```
{embed passage: 'Action!'}

{embed passage: 'Action!'}

{embed passage: 'Action!'}

{embed passage: 'Action!'}
```

Yes, it's the same instruction four times, embedding four instances of the "Action!" scene. Embedding opens or activates the passage, so

we get different content each time. It's as if we had a room containing doors that let us enter varying versions of another room. If this idea isn't working for you, you're not watching enough *Doctor Who*.

Science fiction aside, we can also explain this in terms of programming. As we've seen, embedding is a superpowerful technique that lets us keep all our gnarly code stuff in one place, the better to refine it. If you've worked with object-oriented programming, you'll know the importance of functions or methods, which are bits of code that can be reused (or *invoked*) flexibly as a program operates. An embedded, code-intensive passage does much the same thing as a custom method in JavaScript or Objective C. Chapbook is great for beginners but equally useful for more ambitious coders—which you now are.

We're almost ready to test our project, though we need one slight change. By default, the passage marked as the start of our story is "Action!" because it was created first. While having an embedded passage so marked will not cause your project to break, it also won't put anything on the screen. Select the "Scene" passage, hover your mouse over it, and click on the three dots at the right of the pop-up, which causes a menu to appear. Select "Start Story Here." Now play or test your story.

You should see four links, each displaying a phrase composed by the generator code in "Action!" Click on any link and the current view will refresh with four new links. So far, all we have is another version of our text-generation demos from chapter P-3 or example 5.2—though with perhaps a twist on the content. However, we have one more trick to add.

◊ Close the passage called "Scene" and reopen "Action!" Find the line in the variables section that defines *theRoll*. Change the 6 to a 7. You should have this:

```
theRoll: Math.ceil(random.fraction*7)
```

This is a small change, but important. We're now rolling a (virtual) seven-sided die. Note our use of the *ceil* function to round fraction to integer—this function rounds up to the maximum value, eschewing zero. We'll use this difference for an important feature of the work.

◊ Right after the line you just worked on (the one defining *theRoll*), add the following:

```
escapes: ['Refuse all further action','Prefer
not to','Declare an adventure strike','Stop
suspending disbelief','Stop putting up with this
noise','Have about enough of this']
theEscape: escapes[Math.floor(random.
fraction*escapes.length)]
```

This is one final array-and-selector combination, allowing us to generate a phrase that will be uniquely useful. Meanwhile, at the very bottom of "Action!," below the sixth *if* condition, add a seventh condition as follows:

```
[if theRoll===7]
[[{theEscape}->Escape]]
```

We now have the possibility of throwing a lucky seven. When that happens, the link that will come up in the "Scene" passage will contain one of our "escape" phrases, and it will be linked to a new passage called "Escape."

Close "Action!" and open that new passage. Type anything you like there. We have:

```
And so our story begins for real.
```

You can test the project at this point. When "Scene" comes up, you should see four links. There is a one in seven chance one of these links will contain an "escape" phrase. Clicking on a nonescape link refreshes "Scene" with four more links.

Technically speaking, you have just built a recursive hypertext with the possibility of aleatory termination—look at you! More to the point, you've participated, we're ashamed to admit, in a send-up of the adventure-game idiom. All the nonescape options are based

somewhat loosely on the actions that may be taken in a text adventure. "Take," "Go," "Ask/about"—even "Jump" and "Wait"—are either valid or plausible verbs in systems like TADS or Inform. If one were inclined to make fun of text adventures, insinuating that true story is more like old-fashioned fiction, this might be one way to do it. There's your concept. Whether this is something that needs to be done, we leave to the reader's judgment. If done at all, it must be with affectionate understanding—sibling rivalry or something equally childish. The next time someone shows us a text adventure that makes fun of hypertext fiction, we promise to smile. Yes, Sis, those links are as ridiculous as flared pants.

Of course, it is entirely possible to flip this binary script and write a version of this concept where the real fun lies with the action links—maybe they could do something other than just refresh the screen. The conventional narrative in this version might serve as the obstacle or distraction from the proper story of player action.

Though this chapter is devoted more to concept than technique, we can't resist some technical reflection on the exercise we've just completed. The "Action!" passage uses code far less efficiently than would a comparable structure in Harlowe or JavaScript. As we've seen in chapter P-3, JavaScript permits the use of a structure called *switch* to select one of several phrase templates on each run. Here we have to generate all seven possibilities each time "Action!" is embedded, or twenty-eight times every time "Scene" is accessed.

Why don't we care? Because we're spoiled, twenty-first-century code monkeys who casually toy with machines their grandparents could barely imagine. For us, knocking numbers around is cheap and easy. It's worth remembering that computation implies *heat*—every mathematical operation uses energy—so if playing with Twine makes you interested in learning more ambitious programming, be advised that the world beyond the playground makes harsher demands.

Example 5.4: The Tumblers, or a Tune Out of Season

This project has two inspirations. One is Thomas Pynchon's attribution of "high magic to low puns" (Pynchon 95) The other is Walt Kelly's comic strip *Pogo*, in which we gamboled through postwar America with the laid-back critters of Okefenokee Swamp. Kelly also knew the power of puns: he lampooned the corrosive Spiro T. Agnew as a snorting, fuming "A. Gnu." As we will see, he also had a certain way with holiday songs:

> *Deck us all with Boston Charlie,*
> *Walla Walla, Wash., an' Kalamazoo!*
> *Nora's freezin' on the trolley,*
> *Swaller dollar cauliflower alley-garoo! (Kelly 9)*

Seasonal mondegreenery[1] will figure a bit later in our project. First let's consider the concept, which we admit has more to do with hackery than even midrange magic. As in previous examples, we'll be introducing no new Twine features, relying mainly on the *cycling link* modifier we used in chapter P-3. Keeping that feature in mind, we start with three recognitions:

1. A cycling link is much the same thing as a *tumbler*, the numbered cylinder of a combination lock.
2. In the right context, numbers and words are conveniently interchangeable.
3. A certain low pun comes to mind.

1 A *mondegreen* is a misheard lyric, such as "and Lady Mondegreen" for "and laid him on the green" in "Barbara Allen," or "'scuse me while I kiss this guy" in Jimi Hendrix's "Purple Haze." The actual reading there is "kiss the sky," but what's life without variety? We're also hugely fond of another of Walt Kelly's festive mondegreens, "Good King Sauerkraut look out, on your feets uneven . . ."

The mise-en-scène, or situational framing, is especially important in this example. Let's get to it.

◊ Start a new Twine story called anything you like—for instance, *Tumblers*. Name the default first passage "A Tune Out of Season." In this passage, type the following, or any variation you prefer:

```
Lunch with insufferable Uncle Buster in the
Tumbolia Room. You would do anything to
escape. "Why was Ten glancing so warily up the
numberline?" the old man asks, you suppose you
should say rhetorically, though that's far too
nice a word. "Because . . . ?"
```

Do your best to stifle that groan if you know the punchline. Yes, it's an insufferable-uncle thing. It is also the absurd structure on which we will build our not-so-high concept.

◊ Add a new paragraph to "A Tune Out of Season":

```
Meanwhile, some conventioneers at the bar are
attempting a seasonal song, though it is the
middle of {cycling link for: 'month', choices:
['June', 'July', 'August']}. They have some odd
ideas about the lyrics:
```

We'll get to those lyrics in a bit. First, a review of the *cycling link* modifier. You'll remember it designates a variable—in this case, *month*—for which we supply choices as a comma-separated list enclosed in square brackets. Yes, that's the same form used to define an array because the option list of a cycling link is indeed an array. However, instead of selecting from this array at random, which is our usual method, we'll let Twine do what it does with cycling links, presenting them in sequence. Each time the player clicks the linked month, it will move ahead to the

next of the three options, looping back to June from August. The solution to this puzzle—don't make us say it—consists of three integers. We could offer all twelve months, but the story says the song is out of season, so we're using only the summer months.

◊ Now let's fill in what the chorus is singing in the next room. This time we'll use the full range of options:

```
"{cycling link for: 'song', choices: ['12
mummers slumming', '11 typers griping', '10
Fords a-beeping', '9 maybes branching', '8
trades a-bilking', '7 swamps a-brimming',
'6 tweets a-braying', 'FIVE OLD SPRINGS!',
'4 appalling nerds', '3 clenched pens', '2
hurtful shoves', 'AND A CARTRIDGE WITH A GAME
FREE!']}."
```

Feel free to write your own variations. For the final line, we considered "A SMART FRIDGE WITH SOME SPARE PEAS" and in true *Pogo* idiom, "SOME CARTILAGE FOR YOUR TEARED KNEE." The words matter less than their ordinal position, which is why our convention-goers are singing a sequential carol at the wrong end of the calendar. By now, you may know the final number in our virtual combination, though we have one more alphanumeric trick to play:

◊ Add the following new paragraph:

```
To further unsettle your sanity, the folks at
the next table have started flinging breadsticks
at one another. They are about to proceed to the
cutlery. "Let me put this in a language you will
understand," one of them is saying: "{cycling
link for: 'word', choices: ['MEIYOU','NE','NAY',
'NAE','IIYE','NON','NIX','NEM','NEIN']}!"
```

You can have more or fewer options for *word*, but we're looking for a particular number, *nein*? With the final tumbler in place, we need to make the combination testable.

◊ Add this link at the bottom of "A Tune Out of Season":

```
[[Solution?]]
```

Close the passage and return to the structure map, where you'll find the new passage "Solution?" That's where we'll check out the settings of the three cycling links. (*Three cycling links, two drop-downs, and a . . .* sorry, that's another song.)

◊ In the passage titled "Solution?," enter the following:

```
[if month === 'July' && song === '8 trades
a-bilking' && word === 'NEIN']
"Is that the time?" you notice. "Gosh, late for
that root canal!" And out the door.
```

If our player has set the three cyclers to the correct set of values—really, don't make us say it—our hero gets to flee Uncle Buster and his abominable puns. However, there are 323 possibilities for a wrong answer—$(3 \times 12 \times 9) - 1$—so we need to allow for incorrect solutions.

◊ In the passage titled "Solution?" add the following:

```
[else]
You're stuck here until you solve the stupid
riddle. [[Keep trying->A Tune Out of Season]]

Hint: three numbers in sequence, horrible
pun . . .
```

If you want to be especially kind to your player, you could build an escalating series of hints, like so:

```
[else]
You're stuck here until you solve the stupid
riddle. [[Keep trying->A Tune Out of Season]]

[if passage.visits === 1]
Hint: three numbers in sequence, horrible
pun . . .
[if passage.visits === 2]
Hint: three numbers in sequence, horrible pun,
numerical cannibalism . . .
[if passage.visits === 3]
OK, enough hints: BECAUSE SEVEN ATE NINE!
```

And there it is. This is, of course, an entirely nonserious application of the combination-lock concept, but there's no reason it couldn't be used less foolishly. The options in the cycling links don't have to contain or imply numbers, for instance. The point of the game might be to choose the right sequential settings, presumably in response to some reasonably useful hint, with the cycling options sending the narration or dialogue in meaningful directions. Say we're in a story called *Room 112*, set in a motel, and the combination for each successive passage is the next address along the hallway (114, 116, 118, and so forth), with the correct settings of the cyclers advancing the story, revealing complications among the characters and other things. Or the combinations for successive passages might be permutations of the original, or products of some mathematical operation implied in the text. Or there could be only one cycler in each of many passages, with a final challenge to match some extended series of digits across all of them. Also, of course, it's possible to generate a fresh solution each time the story runs—we leave further word-number permutations to your imagination.

Example 5.5: *Twine Box*

Like the first project in this chapter, this last one has a history, though its origin is more recent than that of the old labyrinth. *Twine Box* was written in the pandemic spring of 2020, just as much of the world was entering lockdown.[2] No surprise that the project is about an enclosed space and the contents of a box. It is also yet another riddle-text. The concept for this project is not simply thematic, though—it is also geometrical. Any box or cube can be flattened into the pattern of a *T*:

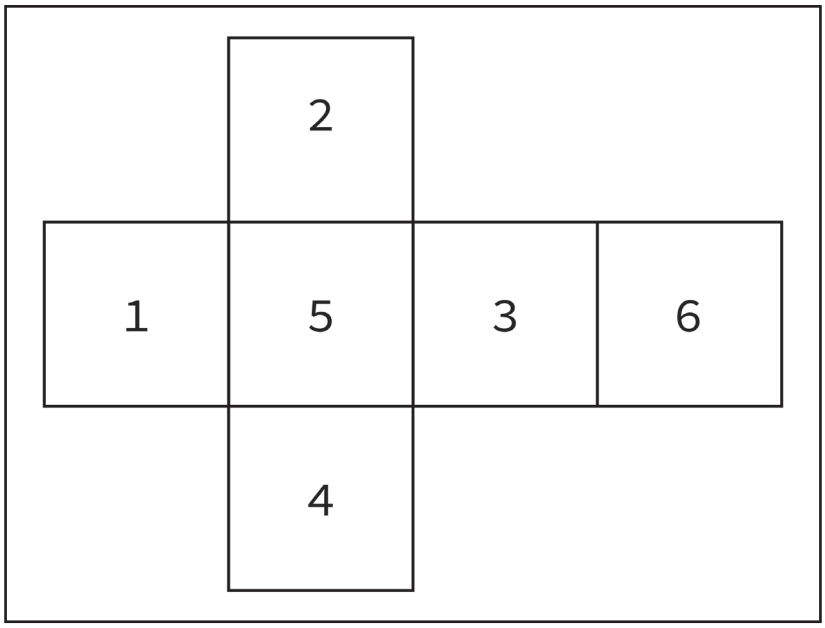

Figure 24: Conceptual diagram of the *Twine Box* project

The six cells represent the sides of the box. You could print this diagram, cut out the image, and fold it into a cube. This image of a deconstructed box led to the architectural scheme for our story: six rooms,

2 The completed version of this project included in the online materials, *Twine Box*, was the basis for a somewhat different story called *Dread Box*, built on the same six-room architecture. It appears in the first issue of *Digital Review*. See http://thedigitalreview.com/issue00/dread-box/begin.html.

each with four transitional links leading to adjacent faces of the cube. You can add the links to the diagram. Starting at the left edge of cell 1 and working clockwise, you might label these links *A* through *D*. Each link has a counterpart on the cell to which it is joined in the cube. Link A connects to the right-hand edge of cell 6, the top of the box. Link B connects to a matching link on the left edge of cell 2. Link C goes on the shared edge with cell 5, and link D, at the bottom of cell 1, matches its partner on the left edge of cell 4. The same logic will define the link pairs in the remaining cells. If you're extending the alphabetic series, as we suggest, your last link pair will be labeled *L*. There are twelve pairs in all.

Moving from geometry to story, let's identify the six rooms as follows:

1 Lecture Hall
2 Committee Room
3 Writers Room
4 Lunch Room
5 Scary Basement
6 Top of the World

These are, of course, fanciful assignments. When it comes to Twine, some of us have a hard time sticking with realism. Through a certain dream-logic, each of these rooms is intended to be a place where our player character can encounter suggestive traces of language—which is to say, clues to the riddle. Let's get started on the build.

◊ Start a new Twine story, making sure your format is set to Chapbook, and name it *Twine Box*, or what you will. Name the default passage "1 The Lecture Hall." Create five more passages and name each according to the aforementioned scheme. In passage "1 The Lecture Hall," enter the following text:

```
Oak paneling, mangled chair-desks, lingering
aura of angst and ennui. At the battered podium,
a vile cigar smokes itself out. The ancient
```

```
chalkboard could use scrubbing. You can make out
some words there:
```

Repeat for each of the five other passages. Here are recommended texts for each:

```
The Committee Room
The door opens just enough to admit the owlish
mug of the deputy assistant secretary. "Private
meeting," she informs you. Inside, you hear
privileged voices saying:

The Writers Room
When you attempt to peek in, an associate
producer rumbles, "GO PLAY IN THE STREET" and
tosses a wad of ill-considered passages toward
your person. "CAN'T YOU SEE WE'VE GOT AN I.F.
TO FINISH HERE?" As the door slams, you hear
unstable voices shouting:

The Lunch Room
For some reason, nobody in here has any clothes
on. Guess we forgot to tell you it's a Naked
Lunchroom! All eyes stare at your overdressed
self. As you back out the door, you hear people
muttering, "In your dreams." Also:

omygosh It's the SCARY BASEMENT!
An ancient bulb flickers feebly and surrenders
to the darkness. The room is filled with uncanny
shapes . . . canopic jars . . . eldritch
apparatuses . . . a cleaner of vacuum. A voice
like an old cigar wails, "BEGONE!" You are about
to do just that when a hovering presence appears,
scrawling words across the skin of reality:
```

```
Top of the World
It's a cross between one of those revolving
restaurants and the bridge of some uncanonical
starship. The reception droid takes your
soulprint and shows you to a mediatronic
terminal. You idly thumb some flashing red
indicators and jettison the city's antimatter
core. Diners at nearby tables murmur:
```

Eventually, there will be text following the colons in each of the six passages, but before we come to that, let's take care of the navigational links.

◊ Open the passage "1 The Lecture Hall." Below the existing text, skip a line and add the following:

```
<div style="text-align: center">
[[3 Writers Room]] [[6 Top of the World]] [[4
Lunch Room]] [[5 Scary Basement]]
</div>
```

There are more elegant ways to arrange these links, but we'll let function win over form for once. Our only concession to formatting is an HTML DIV with centered alignment. We require the player to visit all six rooms, which means you need to be careful about the placement of links, making a full traversal possible. Here are the link sets for the five other passages:

```
2 Committee Room
[[3 Writers Room]] [[5 Scary Basement]] [[4
Lunch Room]] [[6 Top of the World]]

3 Writers Room
[[6 Top of the World]] [[1 The Lecture Hall]]
[[5 Scary Basement]] [[2 Committee Room]]
```

```
4 Lunch Room
[[5 Scary Basement]] [[1 The Lecture Hall]] [[6
Top of the World]] [[2 Committee Room]]

5 Scary Basement
[[3 Writers Room]] [[1 The Lecture Hall]] [[4
Lunch Room]] [[2 Committee Room]]

6 Top of the World
[[3 Writers Room]] [[2 Committee Room]] [[4
Lunch Room]] [[1 The Lecture Hall]]
```

Make the necessary additions to each passage and test your project. You should be able to move from room to room, and if you're keeping track, you should visit all of them eventually. Before we're finished, each room will offer a set of clues about the answer to the riddle, the contents of the conceptual box. In order to ensure exposure to these clues, we want the player to visit every room at least once. When all rooms have been entered, we'll display a new link in each of the rooms to a seventh passage called "INSIDE," representing the interior of the box. Because there's a tedious amount of text involved in the *if* condition for our inside link, we'll set it up as an embedded passage.

◊ Create a new passage and name it "tracker." Enter the following:

```
[if r1 && r2 && r3 && r4 && r5 && r6]
GO [[INSIDE]]
```

We haven't yet created any of those variables starting with *r*—we'll do that in the next step. An explanation about the syntax first. Our r-series variables will be Booleans, with possible values of *true* or *false*. Saying "if *r1*" asks if *r1* has the value *true*. We could write out "if *r1* === true," but that's more typing. Remember, the double ampersands stand

for the logical *and* operator, which means all six subconditions much be true for the main condition to be met.

When you close the "tracker" passage, you will see a new passage called "INSIDE," which we'll leave blank for the moment. First, we need to make sure our r-series variables (*r* stands for *room*) are properly taken care of. We're designing our system so that all six variables need to be checked whenever we enter a room. For that to happen, Twine needs to know about those variables. So far, it doesn't. There's only one proper solution to this problem: create a new starter passage.

◊ Make a new passage and name it "And . . . box." (Yes, there's a mixed metaphor here.) Enter the following into this passage:

```
r1: false
r2: false
r3: false
r4: false
r5: false
r6: false
--
TWINE BOX

[[begin->1 The Lecture Hall]]
```

This is a title passage, with our six tracking variables initially declared and set to *false* in the variables section. Doing this will get us off on the right foot with "tracker."

Next, we have to allow each of our room-tracking variables to become *true*.

◊ Open the passage "1 The Lecture Hall." Insert two blank lines in front of its current contents. Add the following variables section:

```
r1: true
--
```

Make similar changes to the other five room passages, changing the number part of the variable each time (r2 for the second room, r3 for the third, and so on).

Next, we need to embed the "tracker" passage in the link options for each room.

◊ Open the passage "1 The Lecture Hall." In the text body, below the existing series of four links, add the following:

```
{embed passage: 'tracker'}
```

Place this line within the HTML DIV container—before the * * tag. Repeat this procedure for each of the other five room passages. You'll be inserting the same line in the same position for each.

Now the rules of our game are largely implemented. Each time we visit a room, the "tracker" logic will check to see if we have visited all six rooms, in which case it will display the link to "INSIDE." Each passage records its visited state in its *r* variable. You should test your project at this point, visiting each room. When you come to the last unvisited room, you should see the "INSIDE" link.

We'll continue to defer work on that climactic passage because we need to provide the clues we want the player to encounter in each room. In order to make this game minimally playable, we'll pick our clues randomly from large sets, using our substitution grammar. However, we'll want clues of two kinds—right and wrong—so we'll need two sets. Again, we'll handle this feature with embedded passages.

◊ Create a new passage and name it "right." Enter the following:

```
rt: ["thing with feathers","fairey
obama","ancient funnyman bob","and
change","fingers crossed"]

theText: rt[Math.floor(random.fraction*rt.
length)]
--
{theText}
```

The basic scheme here should be entirely familiar: define an array, then a variable that holds one randomly selected item from that array. Both of those things happen in the variables section. In the main text body of this passage, we display the contents of our selector variable. We've included only a small selection from the array used in the finished version of this project. You will want many more than five options for both the "right" and "wrong" clue sets. You can have as many as you want. As for the word those clues indicate, you can probably figure that out, especially if you remember a certain story from Greek mythology. Now let's take care of our not-so-helpful clues.

◊ Create a new passage and name it "wrong." Enter the following:

```
wt: ["seventh of six","if you can read
this","this is not a clue","time fades away","is
time emits I","I is another","you are not
reading this"]

theText: wt[Math.floor(random.fraction*wt.
length)]
--
{theText}
```

The form of this passage is identical to the "right" passage. You'll want to expand the set of options considerably. The wrong-headed

clues can be any misleading or nonsensical expression. Technically, they should not lead to the right solution, though as you'll see, it's not a huge problem if they inadvertently do. Now that we've set up our two clue sets, we need to make use of them. We'll do that in the next steps.

◊ Open passage "1 The Lecture Hall." In the main text body, following the sentence that ends in a colon, skip a line and add the following:

```
{embed passage: 'cluetrain'}
```

Add the same line in the same position in the other five rooms.

Our "cluetrain" passage will take selections from the "right" and "wrong" clue sets, put them in a certain order, and make them ready to appear in each of our rooms. Here's how all that is done:

◊ Create a new passage called "cluetrain"—embedding a passage does not create that passage—and enter the following:

```
<div style="text-indent: 2em">
[if random.coinFlip]
{embed passage: 'wrong'}
[continued]

{embed passage: 'right'}

[if random.coinFlip]
{embed passage: 'wrong'}
[continued]

[if random.coinFlip]
{embed passage: 'wrong'}
[continued]
</div>
```

Inside our familiar HTML DIV container you'll see a series of *embed passage* inserts. Yes, you can embed passages in an embedded passage—just don't embed a passage in itself, as we've already cautioned. The player never suspects how much bed-hopping is going on behind the scenes. In the "cluetrain" passage, the first, third, and fourth embed the "wrong" clue generator. The second embeds the "right" generator. However, we add a coinflip *if* condition to all our wrong-way clues, so they each have a fifty/fifty chance of appearing. Notice we add a *[continued]* modifier to terminate the *if* condition each time. Don't omit that detail.

According to this scheme, our right-leading clue will always appear, though it will sometimes be the first and perhaps only clue. When the coin lands the right way for the initial "wrong" embed, it will be the second clue. This arrangement gives what we consider a minimally interesting amount of variation, though that's a subjective judgment.

It's time to test the project again. You should see between one and four clues following the introductory sentence each time you visit a room. If all is well, you are ready to begin the final stages.

◊ Because it is referenced in a conventional link, the passage called "INSIDE" should already exist. (If it doesn't, go back and check the step in which you set up the link to that passage in the embedded "tracker" passage.) Open "INSIDE" and enter the following:

```
**~~inside the box~~**
So here we are--turning outside in--reaching the
heart of the matter--coming down to core.

Moment of truth . . . or not: _**what's in the
box?**_
```

Those double asterisks and tildes, as well as the underscores around the final phrase, are formatting characters. Lots of dramatic effect here. The payoff is in the selection, for which we'll use a familiar device:

◊ Below what you typed in the previous step, enter the following:

```
{cycling link for: 'answer', choices: ['puzzle',
'nothingness','illusion','ignorance','secrecy','
concealment','hope']}

_**[[REVEAL]]**_
```

Because of the basic scheme of the cycling link, the options will always be presented in the order you code them. The complete version of this project uses *random.d6* to branch among six variations of the cycling link, each with the right answer in a different position. In this version, there are also different wrong options in each version of the cycler. For our purposes, though, let's stay with the simple solution. Now it only remains to test the outcome.

◊ The passage "REVEAL" should have been added to your structure. Open it and enter the following:

```
[if answer==='hope']
Always.
[else]
The box retains its mystery. {restart link,
label: 'Play on'}
```

Besides revealing the answer to the riddle of the "Twine Box" (but you knew), this final piece of code uses a Chapbook feature we first saw in chapter P-2: the *restart link* insert. As the name suggests, a restart link erases all system and custom variables in play, including our *r* series of trackers, and takes us back to our launch passage, which we called "And . . . box." Our six tracking variables are redeclared there, reset to *false*, and we are ready to begin anew.

We've made the design choice to have a fresh start after a wrong guess, requiring the player to receive clues in all six rooms before the "INSIDE" link appears again. If we wanted to be more generous, we

could have avoided the restart link and used an ordinary link back to the lecture hall or any of the other rooms.

And so the secrets of the mystery box have all been revealed, and with that, our tour of conceptual Twine comes to an end. We've dwelt heavily on riddles and puzzles in these five exercises, but as we've seen in preceding chapters, Twine can be used for many more purposes—creative, expressive, analytical, persuasive, and even therapeutic. What you find inside the Twine box, or what you decide to put there, is entirely up to you.

At this point, we're also very near the formal end of *Twining* itself—the conclusion awaits—though as you might expect from such a rambling and rambunctious enterprise, even that final chapter will not really be the end. We invite you to carry on with the appendices, including the interview with Chris Klimas with which this project started and the talk with Dan Cox that guided its growth. For those who still haven't had enough code tinkering, there is even a bonus practical chapter that finds its way "Beyond Twine."

Though if you're like us, you can always find a bit of Twine somewhere around the place.

Works Cited

Kelly, Walt. *Deck Us All with Boston Charlie*. New York: Simon & Schuster, 1963.
Pynchon, Thomas. *The Crying of Lot 49*. Philadelphia: Lippincott, 1966.